University of Twente
The Netherlands

PF/Tijah Documentation

for version 0.3.0 (Pathfinder version 0.20.0)

PF/Tijah (Pathfinder/Tijah, pronounce as "*Pee Ef Teeja*") is a flexible open source text search system developed at the University of Twente in cooperation with CWI Amsterdam and TU München. The system is integrated in the Pathfinder XQuery database system and can be downloaded as part of MonetDB/XQuery.

http://dbappl.cs.utwente.nl/pftijah

Authors: Djoerd Hiemstra, Henning Rode, and Jan Flokstra

Features And Goals

PF/Tijah is a a text search system (Tijah) that is integrated with the Pathfinder (PF) XML database management system. PF/Tijah includes out-of-the-box solutions for common tasks like index creation, document management, stemming, and result ranking (supporting several retrieval models), but it remains the same time open to any adaptation or extension. On the one hand, the system aims to be a general purpose tool for developing IR end-user applications using XQuery statements with text search extensions. On the other hand, the system aims to be a playground for the information retrieval scientist and advanced user to easily set up and test new search systems. Advanced users can hook in the system at an intermediate level that provides the database scripting language MIL and several pre-defined operations on terms and XML elements called Score Region Algebra operators. The PF/Tijah system has a number unique selling points that distinguish it from other information retrieval systems.

- PF/Tijah supports retrieving arbitrary parts of the textual data, unlike traditional information retrieval systems for which the notion of a *document* needs to be defined up front by the application developer. For instance, if the data consist of scientific journals one can query for complete journals, journal issues, single articles, sections from articles or paragraphs with no need to adapt the index or any other part of the system configuration;
- PF/Tijah supports complex scoring and ranking of the retrieved results by means of so-called NEXI queries. NEXI stands for *Narrowed Extended XPath*: a query language that only supports the descendant and the self axis step, but that is extended with a special `about()` function that takes a sequence of nodes and ranks those by their estimated probability of relevance to the query;
- PF/Tijah supports ad hoc result presentation by means of its query language. For instance, when searching for a special issue of a journal, it is easy to print any information from that retrieval result on the screen in a declarative way (i.e., not by means of a general purpose programming language), such as the special issue title, its date, the editors and the preface. This is simply done by means of XQuery element construction;
- PF/Tijah supports text search combined with traditional database querying, including for instance joins on values. For instance, one could search for employees from the financial department that also worked for the sales department and that sent an email about "tax refunds"

Table of Contents

1 Running The Server

PF/Tijah runs on a light database backend: MonetDB. To get started, you have to start the MonetDB server and load the Pathfinder module (of course you first have to install the system, see Download). If your run PF/Tijah under Windows, you start the server via the start menu on the task bar (Start -> MonetDB -> MonetDB XQuery Server). In unix-based systems, open a terminal window and type:

```
Mserver --dbinit="module(pathfinder);"
```

Depending on your hardware and installation, you get get a report similar to the following. (Read the MonetDB4 documentation if the server does not start properly.)

```
# MonetDB Server v4.20.0
# based on GDK   v1.20.0
# Copyright (c) 1993-2007, CWI. All rights reserved.
# Compiled for i686-suse-linux-gnu/32bit with 32bit OIDs; dynamically linked.
# Visit http://monetdb.cwi.nl/ for further information.
# PF/Tijah module v0.3.0 loaded. http://dbappl.cs.utwente.nl/pftijah
# MonetDB/XQuery module v0.20.0 loaded
# XRPC administrative console at http://localhost:50001/admin
```

Warnings like the following: `!WARNING: GDKlockHome: created directory /local/MonetDB/` etc. are normal: the server creates an empty database if you run it for the first time. Let's assume the file `test.xq` contains the following query:

```
<test> { for $i in (1, 2, 3) return $i } </test>
```

Under windows, you can run XQuery statements by double clicking the `test.xq` file, which opens the query results in your web browser. Alternatively, you can open a DOS command prompt window and operate the system as done in Unix-based systems described next: Open *another* terminal window and type:

```
MapiClient -lxquery test.xq
```

resulting in

```
<test>1 2 3</test>
```

You now can go on using the system as described in the Getting started.

2 Getting Started

As XML is mainly a language for document mark-up, *text search* querying or *information retrieval* querying is a natural extension to traditional database querying. Text search querying provides *best matching* instead of *exact matching*, that is, the results of a PF/Tijah text search query do not exactly match the query: Some results may partially match the query. However, the best results are returned first, that is, the results are *not* in document order, but in decreasing order of the estimated probability of relevance to the query. The PF/Tijah extension for text search uses built in XQuery functions and is therefore fully compliant with XQuery 1.0.

Quick Start

To use PF/Tijah search functionality, a full-text index has to be created on a (document) collection. Of course, you first need to start your Mserver (see Running the Server). After loading XML documents into a new collection,

```
let $dir := "/home/pftijah/"
let $files := ("mydata.xml", "moredata.xml", "yetmoredata.xml")
for $file in $files
return pf:add-doc(concat($dir, $file), $file, "MyCollection")
```

the parameter-free call of

```
tijah:create-ft-index()
```

creates such a full-text index on all existing XML collections in the database. The index is automatically updated when new documents are inserted using the standard MonetDB/XQuery document management function `pf:add-doc()`. You are now ready to query the collection. The following XQuery returns the titles of HTML nodes in descending order of their relevance to the query "*ir db*":

```
let $c := collection("MyCollection")
for $res in tijah:query($c, "//html[about(., ir db)]")
return $res/head/title
```

Indexing

A full-text index is always associated with one (or many) MonetDB/XQuery collections, like an index on database table. The index is created by the `tijah:create-ft-index()` function. The following example shows how to create an full-text index on two collections:

```
let $col := ("MyTRECcollection", "MyCLEFcollection")
tijah:create-ft-index($col)
```

Whenever documents are added to one of the associated collections, the full-text index will be updated as well. So you can create the ft-index before or after adding documents to the database. At a later stage, new collections can be associated to an existing index with the function `tijah:extend-ft-index()`:

```
let $col := ("NEWcollection")
tijah:extend-ft-index($col)
```

The indexing can also be parameterized with the several options as described below:

```
let $opt := <TijahOptions stemmer="snowball-english"/>
let $col := ("MyTRECcollection", "MyCLEFcollection")
```

```
tijah:create-ft-index($col, $opt)
```

With the <TijahOptions/> element, the function tijah:create-ft-index() is told to use a specific stemmer, tokenizer, stop words, etc. see Table 1. It is possible to create several indexes, for instance one with stemming and another without. This is specially helpful in scientific evaluations of retrieval approaches. To create multiple indexes, name the index with the TijahOptions attribute ft-index. The ft-index attribute can be used during indexing *and* during querying. At query time the query processor can then be forced to use a certain named full text index.

Attribute	Values	Description
ft-index	string	Name of full-text index that should be used to process the query
stemmer	["**none**", "porter-english", "porter-dutch", "snowball-english", "snowball-dutch"]	Defines the stemmer to be used during indexing

Table 1, TijahOptions for indexing (the bold value is set by default).

MonetDB/XQuery Document management

PF-Tijah builds its full-text indexes on collections (or single documents) stored in a MonetDB/XQuery. MonetDB/XQuery provides built-in functions to persistently store XML documents in the database. Documents are added as follows:

```
pf:add-doc("/home/pftijah/mydata.xml", "mydata.xml")
```

The first argument is the exact path or URL to the document, the second argument is the logical name with which it can be referenced in queries. If you intend to add many small documents, it is wise to group them in MonetDB/XQuery collections as follows. Here, the collection is called *MyCollection*.

```
let $dir := "/home/pftijah/"
let $files := ("mydata.xml", "moredata.xml", "yetmoredata.xml")
for $file in $files
return pf:add-doc(concat($dir, $file), $file, "MyCollection")
```

Text Search

Text search is performed with the built-in function tijah:query(), which inputs a node sequence and a so-called NEXI query. NEXI is an acronym for *Narrowed Extended XPath I*. NEXI queries are 'narrowed XPath' because the only use the descendant step (and self step) from XPath, and they are 'extended XPath' because of the special about() function:

```
let $c := collection("MyCollection")
for $res in tijah:query($c, "//html[about(., ir db)]")
return $res/head/title
```

The function tijah:query() returns a ranked sequence of nodes, that is, the nodes that best match the NEXI query are returned first. Each returned node is scored by PF/Tijah. It is hard to return nodes and scores directly because of the XQuery data model. Therefore, PF/Tijah provides additional built in functions to access nodes and scores separately. To access the scores the query is broken down in a first step: tijah:query-id(), which returns a unique id, and follow-up steps which can be used to return either the nodes with tijah:nodes(), or the scores with tijah:score(), or both as shown below.

```
let $c := collection("MyCollection")
```

```
let $qid := tijah:query-id($c, "//html[about(., ir db)]"))
for $res in tijah:nodes($qid)
let $score := tijah:score($qid, $res)
return <title score="{$score}"> {$res/head/title/text()} </title>
```

The functions are defined such that `tijah:query()` semantically equals `tijah:nodes(tijah:query-id())`, though the latter needs longer to execute.

The system supports two more functions, `tijah:queryall()` and `tijah:queryall-id()`, that search all collections indexed by the full text index. The functions are equal to `tijah:query()` and `tijah:query-id()`, except that they do not take the first parameter of those functions, the start node-set. In case our ft-index is associated only to the collection `MyCollection`, we could abbreviate the introductory example as follows:

```
for $res in tijah:queryall("//html[about(., ir db)]")
return $res/head/title
```

All query functions allow the inclusion of any number of options by means of a special empty XML element `<TijahOptions/>` as the last parameter.

```
let $opt := <TijahOptions returnNumber="10" ir-model="NLLR"/>
let $c := collection("MyCollection")
for $res in tijah:query($c, "//html[about(., xml)]", $opt)
return $res//title
```

Of course, options can also be loaded from a file with `let opt := doc("options.xml")`. With these options, the user can for instance overwrite the default for scoring in an about clause (using the attribute `ir-model`), or specify the number of nodes that need to be returned (using the attribute `returnNumber`). Table 2, lists the options that are currently supported. Please note, some combinations of options might not be supported.

Attribute	**Values**	**Description**
ir-model	["**NLLR**", "LMS", "LM", "OKAPI"]	Specifies the model used to generate scores
collection-lambda	float (between 0 and 1)	Specifies the interpolation parameter for the ir-models "NLLR" and "LMS"
okapi-b	float	Specifies the ir-model "OKAPI" (BM25) parameter b
okapi-k1	float	Specifies the ir-model "OKAPI" (BM25) parameter k1
ft-index	string	Name of full-text index that should be used to process the query
returnNumber	integer	Maximum number of nodes to be returned
rmoverlap	["**false**", "true"]	Remove overlapping elements from the query results
txtmodel_returnall	["**false**", "true"]	If true, then all queried elements (matching and non-matching) are returned
debug	["**0**","1", ... "9"]	Provide different levels of debug information (0 is no debug information)

Table 2, TijahOptions for querying (the bold value is set by default).

Next: See example usages of PF/Tijah in the Show Cases.

3 Show Cases

Here we gather example usage of PF/Tijah, both for research purposes as for rapid development of search applications.

1. Cranfield Demo (How to run a TREC experiment)
2. INEX Demo (A recipe for an INEX experiment)
3. SIGIR Demo (A simple web-based search application)

3.1 Cranfield Demo

The evaluation of information retrieval systems was pioneered in famous experiments conducted at the University of Cranfield by Cyril Cleverdon and his collegues in the late 1950's and early 1960's (Cleverdon 1967). In the Cranfield studies, retrieval experiments were conducted on test databases in a controlled, laboratory-like setting. The aim of the research was to find ways to improve the retrieval effectiveness of information retrieval systems by developing better indexing languages and methods. The components of the Cranfield experiments were:

1. a collection of *documents* (cran.xml),
2. a set of user *requests* (crantop.xml), and
3. a set of *relevance judgments* (cran.qrels), that is a set of documents judged to be relevant to each query

Together, these components form an information retrieval *test collection*. The success of the Cranfield experiments led to researchers arguing for the need of creating an `ideal' test collection (Sparck Jones and Van Rijsbergen 1975). The wish for bigger and more realistic test collections was realised by the Text Retrieval Conferences, TREC for short. The basis of TREC is that a central organisation (the USA National Institute for Standards and Technology: NIST) builds the test collection, and researchers around the world use it to test their own methods and systems. TREC provides common common tasks, common effectiveness measures, and common evaluation procedures (Voorhees and Harman 2005).

How to run a simple TREC experiment

We use the Cranfield data to show how to run a simple TREC experiment. We create a full text index with:

```
tijah:create-ft-index(<TijahOptions stemmer="snowball-english"/>)
<>
```

and we insert the data. We call the collection "CRAN".

```
pf:add-doc("http://dbappl.cs.utwente.nl/pftijah/data/cran.xml",
   "cran.xml", "CRAN")
<>
```

(Please note that <> on the last line is optional. It is used in the interactive terminal to mark the end of a query.) In this case the test data consist of a single XML file, but in collections of more realistic size we will have to add several files. Of course, the Cranfield data at that time was not marked up as XML (or SGML for that matter). We converted the data to XML following the standard markup of most TREC collections. The following query executes Cranfield topic 14 on the data (in TREC, requests are called 'topics').

```
declare function tijah:cran-test001($query_num as xs:integer, $query as xs:string)
  as xs:string*
{
  (: given query number and query, return the top 1000 documents
     in TREC output format :)
  let $opt := <TijahOptions ir-model="NLLR" collection-lambda="0.8"
                            returnNumber="1000"/>
  let $query_text := tijah:tokenize($query)
  let $query_nexi := concat("//DOC[about(., ", $query_text, ")]")
  let $coll := collection("CRAN")
  let $id := tijah:query-id($coll, $query_nexi, $opt)
  for $doc at $rank in tijah:nodes($id)
  return string-join((string($query_num), "Q0",
    normalize-space(exactly-one($doc//DOCNO/text())), string($rank),
      string(tijah:score($id, $doc)), "UT001"), " ")
};
```

```
tijah:cran-test001(14, "papers on shock-sound wave interaction")
<>
```

The user-defined function `tijah:cran-test001()` contains all details of the experiment. The `return` line of the function creates non-XML output for the `trec_eval` program, the standard software provided by TREC to compute effectiveness measures for a run. `trec_eval` needs relational input; it cannot read XML. We use standard XQuery string operations (see: http://www.w3.org/TR/xpath-functions/) to return a sequence of strings that simulates relational output. Pathfinder will format such a sequence as comma-separated strings on separate lines with quotes. For `trec_eval`, quotes and commas need to be removed (see below).

The following query `cran-trec.xq` executes a complete TREC experiment on the Cranfield data. Such an experiment is called a *run* in the remainder of this section. The query creates the output for all requests in the file `topics.xml` .

```
declare function tijah:cran-test001($query_num as xs:integer, $query as xs:string)
  as xs:string*
{
   function as above
};

let $topics_doc := doc("http://dbappl.cs.utwente.nl/pftijah/data/crantop.xml")
for $top in $topics_doc//top
let $query_num := exactly-one($top/num/text())
let $query := exactly-one($top/description/text())
return tijah:cran-test001($query_num, $query)
<>
```

For large collections, running all topics as a single query, i.e., as a single *database transaction*, is unwise, since the whole transaction fails in case of a small error in one of the queries. The transaction will also fail if there is not enough memory to run all 225 Cranfield topics in one transaction.

```
MapiClient -lx cran-trec.xq -odm | tr -d '",' > experiment.out
```

The command `tr -d '",'` removes the quotes and commas (`tr` is a standard Linux command; under Windows, open the output file in an editor and use find-replace to remove quotes and commas): The `trec_eval` program can be downloaded from http://trec.nist.gov/trec_eval . It needs the output run and the cran.qrels file. The evaluation results are generated as follows:

```
trec_eval cran.qrels experiment.out
```

Next: Read how to do an evaluation for the Initiative for the Evaluation of XML Retrieval in: INEX Demo.

References

- Cyril W. Cleverdon (1967). The Cranfield Tests on Index Language Devices, *Aslib Proceedings 19*, pages 173-192. (Reprinted in *Readings in information retrieval*, pages 47 - 59, 1997)
- Karen Sparck Jones, C.J. (Keith) van Rijsbergen (1975). Report on the need for and provision of an" ideal" information retrieval test collection. *British Library Research and Development Report*
- Ellen M. Voorhees and Donna K. Harman, editors (2005). TREC: Experiment and Evaluation in Information Retrieval, *MIT Press*
- Ellen M. Voorhees and Donna K. Harman. Common evaluation measures. In: "Proceedings of the 15th Text Retrieval Conference (TREC), 2006 pdf

3.2 INEX Demo

The INitiative for the Evaluation of XML retrieval (INEX) was set up in 2002 to assist the evaluation of XML search systems. INEX provides several large XML test collections and appropriate scoring methods. Much of the development of the Tijah system was motivated by these collections and methods. This page contains a small recipe for doing INEX with PF/Tijah.

NB The documents, queries and relevance judgements described on this page are available to INEX participants only.

Indexing INEX Wikipedia

INEX 2007 uses a document collection made from English Wikipedia documents. The collection consists of the 4.6 GB of data divided over 659,388 single XML article files, consisting of about 30 million XML elements in total. PF/Tijah is optimized for loading big XML files. The fastest way to insert the data in the database is to glue all tiny XML files together into bigger files, and insert those. This is shown in the following script:

```
#!/bin/sh -e
# Script to index INEX wikipedia. The scripts glues for each part of
# the data the XML files together. The glued files are put in /tmp
# The script assumes that the server can access /tmp, i.e., it runs
# on the same machine
#
MAPI='MapiClient -lx '
echo "tijah:create-ft-index()" | $MAPI
for d in `ls -d part-*`
do
  echo "$d"
  echo "<wiki part=\"$d\">" >/tmp/$d.xml
  find $d -name "*.xml" -exec cat \{\} \; | grep -v "<?xml" >>/tmp/$d.xml
  echo "</wiki>" >>/tmp/$d.xml
  echo "pf:add-doc(\"/tmp/$d.xml\", \"$d.xml\")" | $MAPI
  rm -f /tmp/$d.xml
done
```

This should take about an hour, provided that your machine has enough main memory. If you insist on inserting the files "as they are", you should group them into Pathfinder collections, and you should add them in a small number of transactions (for instance per directory).

Querying INEX Wikipedia

Inex provides topics: small natural language descriptions of a user's information need consisting of a so-called title, a description and a narrative. Additionally, the topic contains a 'castitle', or *content-and-structure* title: a structured information retrieval query using NEXI. NEXI stands for *Narrowed Extended XPath I*, a narrowed version of XPath that uses descendant steps only, and that is extended with a special `about()` function.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE inex_topic SYSTEM "topic.dtd">
<inex_topic topic_id="405"  ct_no="197">
<title>"The Old Man and the Sea"</title>
<castitle>//*[about(.//collectionlink,"the old man and the sea")]</castitle>
<description>Looking for references to the book "The Old Man and the Sea"</description>
 <narrative>I am writing an essay on the book "The Old Man and the Sea" and would
  like to get all the contexts in which the book is referenced.</narrative>
<ontopic_keywords>Old Man Sea Hemingway Ernest</ontopic_keywords>
</inex_topic>
```

The query below produces the output for INEX topic 289. The query starts of with a function for generating the paths of the result elements. The query gets the nexi query from the topics file 289.xml, executes the query and gathers all results to produce the official INEX run result. For more information, see the: INEX 2006 Retrieval Task and Result Submission Specification

```
(: INEX query for topic 289 written by Roel van Os :)
declare function tijah:getINEXPath ( $node as node() ) as xs:string
{
  (: This function determines for a given XML element the path to it from
     the article root. This path expression is a valid XPathâexpression,
     in the form /article[1]/sec[1]/p[3]
    :)
  let $pathelements := for $a in $node/ancestor-or-self::*
    where ( name($a) != "books" and name($a) != "journal"  (: for IEEE collection :)
          and name($a) != "wiki" )
    return
      if ( name($a) = "article" ) then "article[1]"
      else concat(name($a), "[" ,
        count($a/preceding-sibling::*[name()=name($a)]) + 1, "]")
  return string-join( $pathelements, "/" )
};

(: Configuration of the IR subsystem. Many more options apply,
   see Documentation :)
let $options := <TijahOptions returnNumber="1500"/>
let $topic := doc("/local/inex/topics-2006-005/289.xml")//inex_topic
let $nexi := string-join($topic/castitle/text(),"")

(: Perform the IR query, returning a sequence of nodes :)
let $nodes := tijah:queryall($nexi, $options)

(: Return a topic element :)
return <topic topic-id="{$topic/@topic_id}"> {
  for $node at $rank in $nodes
  return
    <result>
      <file> { data($node/ancestor-or-self::article/name/@id) } </file>
      <path> { tijah:getINEXPath($node) } </path>
      <rank> { $rank } </rank>
    </result>
  } </topic>
```

To get evaluation results for a full INEX run, you need the INEX *relevance judgements* and the EvalJ evaluation program.

Next: Read how to make a simple web-based search application in the SIGIR Demo.

3.3 SIGIR Demo

The SIGIR demo can be found here: http://www.sigir2007.org/search

PF/Tijah is perfect for rapid prototyping and for quickly developing search applications. This page describes the use of XQuery for building a web-based search engine. We describe the SIGIR search demo developed for the 30th anniversary of SIGIR. The demo system searches the abstracts the SIGIR proceedings of the last 30 years: (sigir.xml). Below you see an excerpt of the data.

```
<proceedings>
  <mtitle>SIGIR 2004</mtitle>
  <paper>
    <authors>
      <author>Djoerd Hiemstra</author>
      <author>Stephen E. Robertson</author>
      <author>Hugo Zaragoza</author>
    </authors>
    <locations>
      <location>The Netherlands</location>
      <location>UK</location>
    </locations>
    <title>Parsimonious language models for information retrieval</title>
    <abstract>
     We systematically investigate a new approach to estimating the parameters
     of language models for information retrieval, called parsimonious
     language models. Parsimonious language models explicitly address the
     relation between levels of language models thatare typically used for
     smoothing. As such, they need fewer (non-zero) parameters to describe
     the data.
(...)
```

Inserting the data

As before, the data is inserted in the database by creating the index, and adding one or more documents. In the following query, we combined index creation and document insertion in one single database transaction. We named the index "SIGIR" and use Porter's stemmer for English. (<> at the end is only needed in the interactive terminal to mark the end of the query.)

```
let $opt := <TijahOptions ft-index="SIGIR" stemmer="snowball-english"/>
return
( tijah:create-ft-index($opt),
  pf:add-doc("http://dbappl.cs.utwente.nl/pftijah/data/sigir.xml", "sigir.xml")
)
<>
```

Querying the data

The demo shows several search applications that return elements that do not themselves contain the content searched for: expert search, temporal search and geographical search. For instance, when searching for an expert on *"XML retrieval"*, the author element itself does not contain the word *XML"* nor the word *"retrieval"*. For each distinct author, therefore, we have to aggregate (in this case: sum up) the scores of papers they authored, and order the authors by the aggregated scores. The top author can be considered the SIGIR expert on XML retrieval. This is done in the following query.

```
let $doc := doc("sigir.xml")
let $nexi := "//paper[about(.//title, XML retrieval) or
                      about(.//abstract, XML retrieval)]"
let $opt := <TijahOptions ft-index="SIGIR" ir-model="LMS"
   collection-lambda="0.8" returnNumber="50" />
```

```
let $qid := tijah:query-id($doc, $nexi, $opt)
let $nodes := tijah:nodes($qid)
let $result :=
  for $author in distinct-values ($nodes//author) (: for every distinct author :)
  let $papers := for $paper in $nodes
                 where $paper//author = $author
                 return $paper                    (: get his/her papers :)
  let $score := sum( for $paper in $papers
                     return tijah:score($qid, $paper) div
                         count($paper//author) )  (: sum up the scores per author :)
  order by $score descending
  return
  <expert tijahscore="{$score}"> <name> { $author } </name>
  { for $p at $r in $papers
    where $r < 5
    return
    <evidence> { $p//title }
      <authors> { string-join($p//author, ", ") } </authors>
      <booktitle>Proceedings of { $p/../mtitle/text() } </booktitle>
    </evidence>
  }
  </expert>
(: to get the first 10 :)
for $res at $rank in subsequence($result, 1, 10)
return $res
<>
```

The output of this query contains XML elements like `<expert>` and `<evidence>` that have to be translated by the web application to HTML tags. In a real search application, the user must be able to inspect the abstracts of the returned documents, the full proceedings in which they appeared, and the other SIGIR papers written by the expert. Since PF/Tijah is embedded in MonetDB/XQuery, this information is easily selected from the database by ordinary queries. For instance, selecting all SIGIR papers authored by *"Norbert Fuhr"*, is done as follows:

```
for $p in doc("sigir.xml")//paper[.//author = "Norbert Fuhr"]
return $p
<>
```

Using CGI and the MapiClient

Have a look at the following basic Perl CGI script, sigirtest.cgi, for a simple but complete example implementation of the SIGIR expert search application. The script is meant to run on an Apache Web server. Please note that the script can be improved in several ways (for instance by proper use of the Perl modules like CGI.pm). The exact same script is running here: SIGIR demo test version.

Using XML RPC

MonetDB provides a standard interface for XML Remote Procedure Calls (XML RPC), which allows calling simple remote XQuery functions such as `sigir:ExpertSearch()` as defined in the following module that is stored on the server: http://pathfinder.cs.utwente.nl:54775/export/sigir.xq (*for security reasons we have closed access from some ip numbers, drop us a message if you like to get access, or download the XRPC example method here: sigir.xq*) The function `sigir:ExpertSearch()` acts as a stored procedure that can be called from a client programme using XML RPC. Have a look at the following simple Java class TijahSoap.java and the Java programme ExpertSearch.java for an example of how to call the expert search function via XML RPC in Java. Please note that these Java examples can be improved in several ways again, for instance using standard classes for XML RPC instead of simply passing the XML RPC call via HTTP to the server. More information about XRPC in MonetDB can be found in the MonetDB XQuery documentation.

4 Install From CVS

The development is done on the pathfinder CVS. To install the CVS version of pathfinder (on a Linux or other Unix-based system), download and run the following script: install-pftijah-cvs.sh, change it's MONETDB_PREFIX variable to point to the install directory of your preference. The script follows the steps explained below.

Requirements

You need the following tools to compile MonetDB and pathfinder from CVS:

- make
- a C/C++-compiler. gcc is recommended.
- autoconf
- automake
- autogen
- libtool
- flex (or equivalent)
- bison (or equivalent)
- python, including headers and python-devel
- libxml2, including headers

You first have to decide where your MonetDB installation should reside. You can install it locally e.g. in a subdirectory of your home directory, or you can install it globally (system-wide). In the case of a system-wide installation, no extra steps have to be performed: you can skip step 1 and proceed directly to step 2.

Step 1 (local installation): prepare directories and search paths

In case of a local installation you have to prepare this directory before installing. Suppose you are using `/local/MonetDB` as the installation directory:

```
export MONETDB_PREFIX=/local/MonetDB
mkdir $MONETDB_PREFIX
```

Of course, the latter command is only necessary if the directory did not exist yet. In the remainder of this page, *`$MONETDB_PREFIX`* signifies the installation directory you chose. The following can be placed at the end of your shell's configuration file (e.g. `~/.bashrc`):

```
PATH=$PATH:$MONETDB_PREFIX/bin
export PATH
```

Step 2: checkout, compile and install

Pathfinder/Tijah is split into several modules. The ones of interest to us are:

- *buildtools*: these tools are required to compile MonetDB and pathfinder from CVS.
- *MonetDB*: this is the database kernel, with only the common functionality shared by the different front-ends.
- *clients*: contains among others the MapiClient
- *MonetDB4*: contains version 4 of the database server
- *pathfinder*: contains the XQuery frontend to MonetDB. This includes the PF/Tijah extension.

To checkout, compile and install PF/Tijah, you have to perform the following steps for each of the modules above, **in the order listed**. The order is important because for example the clients module depends on the MonetDB module.

```
cvs -d:pserver:anonymous@monetdb.cvs.sourceforge.net:/cvsroot/monetdb \
  checkout modulename
cd modulename
./bootstrap
./configure --prefix=$MONETDB_PREFIX  --enable-pftijah=yes
     [--enable-bits=64] [--enable-oid32]
make install
```

Notes about `configure`:

- If you are performing a system-wide installation, you can omit the `--prefix` argument.
- Use `--enable-bits=64` in case you are compiling on a 64-bit machine. You might then also add `--enable-oid32` to use 32-bit object identifiers (oids). This is recommended if you don't need the scalability of 64-bit object identifiers: these take up a lot of disk space and memory.

Important: charming the python

After installing the buildtools, but *before* installing the other packages, you have to set the `$PYTHONPATH` environment variable:

```
export PYTHONPATH=`find $MONETDB_PREFIX/lib*/python* -type d -name "site-packages"`
```

Using a release branch instead of the development branch

The cvs command above checks out the development version of all the modules. This development version is often unstable. To get a reasonably stable working installation, you can check out the most recent stable branch, which receives bugfixes only. To do this, pass the following tag names using the `-r` option to CVS:

- *buildtools*: MonetDB_1-20
- *MonetDB*: MonetDB_1-20
- *clients*: Clients_1-20
- *MonetDB4*: MonetDB_4-20
- *pathfinder*: XQuery_0-20

So, the command to check out the stable version of the buildtools becomes:

```
cvs -d:pserver:anonymous@monetdb.cvs.sourceforge.net:/cvsroot/monetdb \
  checkout -r MonetDB_1-20 buildtools
```

After installing the system, you should proceed with Running the server

5 Quick Reference

Table of contents

---

pf:add-doc

Signatures: `pf:add-doc($url as xs:string, $name as xs:string) as docmgmt`
`pf:add-doc($url as xs:string, $name as xs:string, $coll as xs:string) as docmgmt`

Purpose: Add a document to the database

Parameters: `$url`: full path or url of the XML document
`$name`: logical name of the document in the database (must be unique)
`$coll`: name of collection to which the document is added *(optional)*

Returns: Nothing (`docmgmt` is an internal type that is returned by document management functions. Items of type `docmgmt` are not serialized as XML)

Status: IMPLEMENTED

Remarks: If no collection is specified, the logical name of the document will function as a collection with one document.

---

tijah:create-ft-index

Signatures: `tijah:create-ft-index( ) as docmgmt`
`tijah:create-ft-index( $colls as xs:string* ) as docmgmt`
`tijah:create-ft-index( $opt as node() ) as docmgmt`
`tijah:create-ft-index( $colls as xs:string*, $opt as node() ) as docmgmt`

Purpose: Create a full text index on one or more Pathfinder collections.

Parameters: `$colls`: Sequence of collection names to index. *(optional)*
`$opt`: A single node `<TijahOptions />` *(optional)*

Returns: Nothing (`docmgmt` is an internal type that is returned by document management functions. Items of type `docmgmt` are not serialized as XML)

Status: IMPLEMENTED

Remarks: If no collections are specified, the full text index is created for all collections in the database.

---

tijah:delete-ft-index

Signatures: `tijah:delete-ft-index( ) as docmgmt`
`tijah:delete-ft-index( $opt as node() ) as docmgmt`

Purpose: Delete a full text index.

Parameters: `$opt`: A single node `<TijahOptions />` *(optional)*

Returns: Nothing (see `tijah:create-ft-index()`)

Status: IMPLEMENTED

Remarks: The option node is only needed to remove named indexes, for instance `<TijahOptions ft-index="myindex"/>`.

---

tijah:extend-ft-index

Signatures: `tijah:extend-ft-index( $colls as xs:string* ) as docmgmt`
`tijah:extend-ft-index( $colls as xs:string*, $opt as node() ) as docmgmt`

Purpose: Extend a full text index with one or more Pathfinder collections.

Parameters: `$colls`: Sequence of collection names to index. *(optional)*
`$opt`: A single node `<TijahOptions />` *(optional)*

Returns: Nothing (see `tijah:create-ft-index()`)

Status: IMPLEMENTED

Remarks: The option node is only needed to extend named indexes, for instance `<TijahOptions ft-index="myindex"/>`.

---

tijah:ft-index-info

Signatures: `tijah:ft-index-info( ) as element()*`
`tijah:ft-index-info( $ft-index as xs:string ) as element()*`

Purpose: Give info about existing full text indexes.

Parameters: `$ft-index`: Name of full text index *(optional)*

Returns: A single element for each index, for instance `<ftindex collections="*" tokenizer="flex" stemmer="nostemming">DFLT_FT_INDEX</ftindex>`, or nothing if there is no index.

Status: IMPLEMENTED

---

tijah:nodes

Signature: `tijah:nodes( $qid as xs:integer ) as node()*`

Purpose: Get the result nodes given a query result identifier.

Parameters: `$qid`: the query identifier must be created by `tijah:query-id()` or `tijah:queryall-id()` in the same XQuery statement.

Returns: A sequence containing the result nodes sorted on relevance.

Status: IMPLEMENTED

---

tijah:query

Signatures: `tijah:query( $start as node()*, $nexi as xs:string ) as node()*`
`tijah:query( $start as node()*, $nexi as xs:string, $opt as node() ) as node()*`
Purpose: Run an NEXI query on a predefined full-text index.
Parameters: `$start`: Sequence of nodes to restrict the query on.
`$nexi`: NEXI query.
`$opt`: A single node `<TijahOptions />` *(optional)*
Returns: A sequence of nodes, sorted on relevance
Status: IMPLEMENTED
Remarks: Semantically, this function is a shortcut for `tijah:nodes( tijah:query-id() )`.

---

tijah:query-id

Signature: `tijah:query-id( $start as node()*, $nexi as xs:string ) as xs:integer`
`tijah:query-id( $start as node()*, $nexi as xs:string, $opt as node() ) as xs:integer`
Purpose: Run an NEXI query on a predefined full-text index.
Parameters: `$start`: Sequence of nodes to restrict the query on.
`$nexi`: NEXI query.
`$opt`: A single node `<TijahOptions />` *(optional)*
Returns: A unique query result identifier.
Status: IMPLEMENTED

---

tijah:queryall

Signatures: `tijah:query( $nexi as xs:string ) as node()*`
`tijah:query( $nexi as xs:string, $opt as node() ) as node()*`
Purpose: Run an NEXI query on all data in a predefined full-text index.
Parameters: `$nexi`: NEXI query.
`$opt`: A single node `<TijahOptions />` *(optional)*
Returns: A sequence of nodes, sorted on relevance
Status: IMPLEMENTED
Remarks: Semantically, this function is a shortcut for `tijah:nodes( tijah:queryall-id() )`.

---

tijah:queryall-id

Signature: `tijah:queryall-id( $nexi as xs:string ) as xs:integer`
`tijah:queryall-id( $nexi as xs:string, $opt as node() ) as xs:integer`
Purpose: Run an NEXI query on a predefined full-text index.
Parameters: `$nexi`: NEXI query.
`$opt`: A single node `<TijahOptions />` *(optional)*
Returns: A unique query result identifier.

Status: IMPLEMENTED

---

tijah:resultsize

Signature: `tijah:resultsize( $qid as xs:integer ) as xs:integer`

Purpose: Return the size of the result of a tijah query.

Parameters: `$qid`: the query identifier must be created by `tijah:query-id()` or `tijah:queryall-id()` in the same XQuery statement.

Returns: Number of nodes to be returned by the query.

Status: IMPLEMENTED

---

tijah:score

Signature: `tijah:score( $qid as xs:integer, $node as node()* ) as xs:double*`

Purpose: Get the score values from a sequence of nodes.

Parameters: `$qid`: the query identifier must be created by `tijah:query-id()` or `tijah:queryall-id()` in the same XQuery statement.

`$node`: sequence of nodes

Returns: A sequence containing the scores of the nodes passed in the second argument, in the same order. If a node was not scored it gets a score zero.

Status: IMPLEMENTED

---

tijah:tokenize

Signature: `tijah:tokenize( $str as xs:string ) as xs:string`

Purpose: Tokenize a string using the standard Tijah tokenizer.

Parameters: `$str`: The string to be tokenized.

Returns: The tokenized string

Status: IMPLEMENTED
